Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators

Yunsup Lee[*], Rimas Avizienis[*], Alex Bishara[*], Richard Xia[*], Derek Lockhart[†], Christopher Batten[†], and Krste Asanović[*]

[*]Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA
{yunsup,rimas,abishara,rxia,krste}@eecs.berkeley.edu

[†]School of Electrical and Computer Engineering
Cornell University, Ithaca, NY
{dml257,cbatten}@cornell.edu

ABSTRACT

We present a taxonomy and modular implementation approach for data-parallel accelerators, including the MIMD, vector-SIMD, subword-SIMD, SIMT, and vector-thread (VT) architectural design patterns. We have developed a new VT microarchitecture, Maven, based on the traditional vector-SIMD microarchitecture that is considerably simpler to implement and easier to program than previous VT designs. Using an extensive design-space exploration of full VLSI implementations of many accelerator design points, we evaluate the varying tradeoffs between programmability and implementation efficiency among the MIMD, vector-SIMD, and VT patterns on a workload of microbenchmarks and compiled application kernels. We find the vector cores provide greater efficiency than the MIMD cores, even on fairly irregular kernels. Our results suggest that the Maven VT microarchitecture is superior to the traditional vector-SIMD architecture, providing both greater efficiency and easier programmability.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*array and vector processors, MIMD, SIMD*

General Terms

Design

1. INTRODUCTION

Data-parallel kernels dominate the computational workload in a wide variety of demanding application domains, including graphics rendering, computer vision, audio processing, physical simulation, and machine learning. Specialized data-parallel accelerators [6, 8, 10, 16, 22] have long been known to provide greater energy and area efficiency than general-purpose processors for codes with significant amounts of data-level parallelism (DLP). With continuing improvements in transistor density and an increasing emphasis on energy efficiency, there has recently been growing interest in DLP accelerators for mainstream computing environments. These accelerators are usually attached to a general-purpose host processor, either on the same die or a separate die. The host processor executes system code and non-DLP application code while distributing DLP kernels to the accelerator. Surveying the wide range of data-parallel accelerator cores in industry and academia reveals a general tradeoff between programmability (how easy is it to write software for the accelerator?) and efficiency (energy/task and tasks/second/area). In this paper, we examine multiple alternative data-parallel accelerators to quantify the efficiency impact of microarchitectural features intended to simplify programming or expand the range of code that can be executed.

We first introduce a set of five architectural design patterns for DLP cores in Section 2, qualitatively comparing their expected programmability and efficiency. The MIMD pattern [8] flexibly supports mapping data-parallel tasks to a collection of simple scalar or multithreaded cores, but lacks mechanisms for efficient execution of regular DLP. The vector-SIMD [19, 22] and subword-SIMD [6] patterns can significantly reduce the energy on regular DLP, but can require complicated programming for irregular DLP. The single-instruction multiple-thread (SIMT) [12] and vector-thread (VT) [10] patterns are hybrids between the MIMD and vector-SIMD patterns that attempt to offer alternative tradeoffs between programmability and efficiency.

When reducing these high-level patterns to an efficient VLSI design, there is a large design space to explore. In Section 3, we present a common set of parameterized synthesizable microarchitectural components and show how these can be combined to form complete RTL designs for the different architectural design patterns, thereby reducing total design effort and allowing a fairer comparison across patterns. In this section, we also introduce Maven, a new VT microarchitecture. Our modular design strategy revealed a much simpler and more efficient implementation than the earlier Scale VT design [9, 10]. Maven [2, 11] is based on a vector-SIMD microarchitecture with only the minimum number of hardware mechanisms added to enable the improved programmability from VT, instead of the decoupled cluster microarchitecture of Scale. Another innovation in Maven is to use the same RISC ISA for both vector and scalar code, greatly reducing the effort required to develop an efficient VT compiler. The Scale design required a separate clustered ISA for vector code, which complicated compiler development [7].

To concretely evaluate and compare the efficiency of these patterns, we have generated and analyzed hundreds of complete VLSI layouts for the MIMD, vector-SIMD, and VT patterns using our parameterized microarchitecture components targeting a modern 65 nm technology. Sections 4 describes our methodology for ex-

ISCA'11, June 4–8, 2011, San Jose, California, USA.

tracting area, energy, and performance numbers for a range of microbenchmarks and compiled application kernels. Section 5 presents and analyzes our results.

Our results show that vector cores are considerably more efficient in both energy and area-normalized performance than MIMD cores, although the MIMD cores are usually easier to program. Our results also suggest that the Maven VT microarchitecture is superior to the traditional vector-SIMD architecture, providing greater efficiency and a simpler programming model. For both VT and vector-SIMD, multi-lane implementations are usually more efficient than multi-core single-lane implementations and can be easier to program as they require less partitioning and load balancing. Although we do not implement a SIMT machine, some initial analysis indicates SIMT will be less efficient than VT but should be easier to program.

2. ARCHITECTURAL DESIGN PATTERNS

We begin by categorizing kernels encountered in data-parallel applications, which usually include a mix of regular and irregular DLP [10, 13, 18, 20] as illustrated in Figure 1. *Regular DLP* has well-structured data accesses with regular address streams that are known well in advance and includes well-structured control flow. *Irregular DLP* has less-structured data accesses with dynamic and difficult to predict address streams, and also has less-structured data-dependent control flow. Extremely irregular DLP is probably better categorized as task-level parallelism. Accelerators that handle a wider variety of DLP are more attractive for many reasons. First, it is possible to improve efficiency even on irregular DLP. Second, even if efficiency on irregular DLP is similar to a general-purpose processor, keeping the work on the accelerator makes it easier to exploit regular DLP intermingled with irregular DLP. Finally, a consistent way of mapping regular and irregular DLP simplifies programming. The rest of this section presents five architectural patterns for the design of data-parallel accelerators, and describes how each pattern handles both regular and irregular DLP.

The **MIMD** pattern includes a large number of scalar or multi-threaded cores, and one or more data-parallel tasks are mapped to each core. Figure 2(a) shows the programmer's logical view and a typical microarchitecture for this pattern. All design patterns include a *host thread* (HT) that runs on the general-purpose processor and is responsible for application startup, configuration, interaction with the operating system, and managing the accelerator. We refer to the threads that run on the accelerator as *microthreads* (μTs), since they are lighter weight than the host threads. The primary advantage of the MIMD pattern is its simple and flexible programming model, with little difficulty in mapping both regular and irregular DLP. The primary disadvantage is that MIMD does not exploit DLP to improve area and energy efficiency. The pseudo-assembly in Figure 3(a) illustrates how we might map a portion of a simple irregular loop to each μT. An example of the MIMD pattern is the recently proposed 1000-core Rigel accelerator [8], with a single μT per scalar core.

In the **vector-SIMD** pattern a *control thread* (CT) executes vector memory instructions to move data between memory and vector registers, and vector arithmetic instructions to operate on vectors in registers. As shown in Figure 2(b), each CT manages an array of μTs that execute as if in lock-step; each μT is responsible for one element of the vector and the hardware vector length is the number of μTs. The HT allocates work at a coarse-grain to the CTs, and the CT in turn distributes work to the μTs with vector instructions, enabling very efficient execution of fine-grain DLP. Typically, the CT

```
for ( i = 0; i < n; i++ )
  C[i] = x * A[i] + B[2*i];
```

(a) Regular DA, Regular CF

```
for ( i = 0; i < n; i++ )
  E[C[i]] = D[A[i]] + B[i];
```

(b) Irregular DA, Regular CF

```
for ( i = 0; i < n; i++ )
  x = ( A[i] > 0 ) ? y : z;
  C[i] = x * A[i] + B[i];
```

(c) Regular DA, Irregular CF

```
for ( i = 0; i < n; i++ )
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
```

(d) Irregular DA, Irregular CF

```
for ( i = 0; i < n; i++ )
  C[i] = false; j = 0;
  while ( !C[i] & (j < m) )
    if ( A[i] == B[j++] )
      C[i] = true;
```

(e) Irregular DA, Irregular CF

Figure 1: Different Types of Data-Level Parallelism – Examples expressed in a C-like pseudocode and are ordered from regular to irregular DLP. DA = data access, CF = control flow.

is mapped to a *control processor* (CP) and the μTs are mapped spatially and/or temporally across one or more *vector lanes* in the vector unit. The *vector memory unit* (VMU) executes vector memory instructions, and the *vector issue unit* (VIU) handles hazard checking and dispatch of vector arithmetic instructions. Figure 3(b) illustrate three ways vector-SIMD improves energy efficiency: (1) some instructions are executed once by the CT instead of once for each μT (inst. 1, 10–14); (2) for operations that μTs do execute (inst. 4–9), the CP and VIU can amortize overheads (instruction fetch, decode, hazard checking, dispatch) over `vlen` elements; and (3) for μT memory accesses (inst. 4–5, 9), the VMU can efficiently move data in large blocks. Mapping regular DLP to the vector-SIMD pattern is relatively straightforward. Irregular DLP requires the use of vector flags to implement data-dependent conditional control flow, as shown in Figure 3(b). More complicated irregular DLP with nested conditionals can quickly require many independent flag registers and complicated flag arithmetic [21]. The T0 vector microprocessor [22] is an example of this pattern.

The **subword-SIMD** pattern, shown in Figure 2(c), uses wide scalar registers and datapaths (often overlain on a double-precision floating-point unit) to provide a "vector-like" unit. Unlike the vector-SIMD pattern, subword-SIMD accelerators usually have fixed-length vectors, memory alignment constraints, and limited support for irregular DLP. The IBM Cell [6] exemplifies this pattern with a subword-SIMD unit supporting scalar operations as well as 16×8-bit, 8×16-bit, 4×32-bit, and 2×64-bit vector operations. In this work, we do not further consider the subword-SIMD pattern, as the vector-SIMD pattern is better suited to exploiting large amounts of regular and irregular DLP.

The **SIMT** pattern is a hybrid combining the MIMD pattern's logical view with the vector-SIMD pattern's microarchitecture. As shown in Figure 2(d), the SIMT pattern has no CTs; the HT is responsible for directly managing the μTs, usually in *blocks* of μTs. The VIU executes multiple μTs' scalar instructions using SIMD execution as long as they proceed along the same control path. Unlike vector-SIMD, which has a separate CT, the μTs must redundantly execute some instructions (inst. 1–2, 5–7 in Figure 3(c)) and regular data accesses must be encoded as multiple scalar accesses (inst. 3, 8, 11), although these can then be dynamically coalesced, at some area/energy overhead, into vector-like memory operations. The lack of a CT also requires per μT stripmining calculations (inst. 1). The real benefit of SIMT, however, is that it provides a

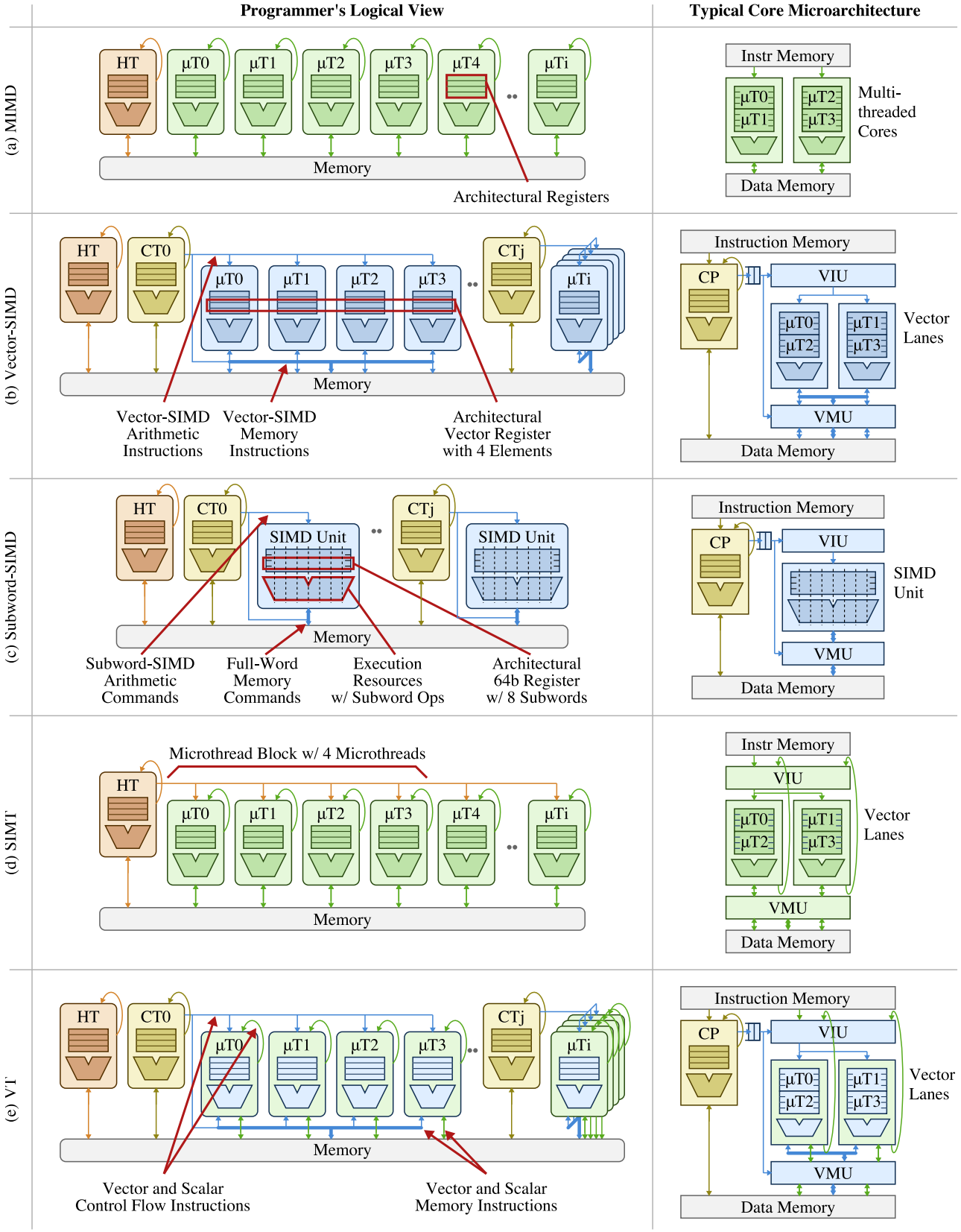


Figure 2: Architectural Design Patterns – Programmer's logical view and a typical core microarchitecture for five patterns: (a) MIMD, (b) vector-SIMD, (c) subword-SIMD, (d) SIMT, and (e) VT. HT = host thread, CT = control thread, CP = control processor, µT = microthread, VIU = vector issue unit, VMU = vector memory unit.

```
1   div    m, n, nthr
2   mul    t, m, tidx
3   add    a_ptr, t
4   add    b_ptr, t
5   add    c_ptr, t
6   sub    t, nthr, 1
7   br.neq t, tidx, ex
8   rem    m, n, nthr
9 ex:
10  load   x, x_ptr
11 loop:
12  load   a, a_ptr
13  br.eq  a, 0, done
14  load   b, b_ptr
15  mul    t, x, a
16  add    c, t, b
17  store  c, c_ptr
18 done:
19  add    a_ptr, 1
20  add    b_ptr, 1
21  add    c_ptr, 1
22  sub    m, 1
23  br.neq m, 0, loop
```

(a) MIMD

```
1   load     x, x_ptr
2 loop:
3   setvl    vlen, n
4   load.v   VA, a_ptr
5   load.v   VB, b_ptr
6   cmp.gt.v VF, VA, 0
7   mul.sv   VT, x, VA,  VF
8   add.vv   VC, VT, VB, VF
9   store.v  VC, c_ptr,  VF
10  add      a_ptr, vlen
11  add      b_ptr, vlen
12  add      c_ptr, vlen
13  sub      n, vlen
14  br.neq   n, 0, loop
```

(b) Vector-SIMD

```
1   br.gte tidx, n, done
2   add    a_ptr, tidx
3   load   a, a_ptr
4   br.eq  a, 0, done
5   add    b_ptr, tidx
6   add    c_ptr, tidx
7   load   x, x_ptr
8   load   b, b_ptr
9   mul    t, x, a
10  add    c, t, b
11  store  c, c_ptr
12 done:
```

(c) SIMT

```
1   load    x, x_ptr
2   mov.sv  VZ, x
3 loop:
4   setvl   vlen, n
5   load.v  VA, a_ptr
6   load.v  VB, b_ptr
7   mov.sv  VD, c_ptr
8   fetch.v ut_code
9   add     a_ptr, vlen
10  add     b_ptr, vlen
11  add     c_ptr, vlen
12  sub     n, vlen
13  br.neq  n, 0, loop
14  ...
15 ut_code:
16  br.eq   a, 0, done
17  mul     t, z, a
18  add     c, t, b
19  add     d, tidx
20  store   c, d
21 done:
22  stop
```

(d) VT

Figure 3: Pseudo-Assembly for Irregular DLP Example – Pseudo-assembly implements the loop in Figure 1(d) for the (a) MIMD, (b) vector-SIMD, (c) SIMT, and (d) VT patterns. Assume `*_ptr` and `n` are inputs. `V`*i* = vector register *i*, `VF` = vector flag register, `*.v` = vector command, `*.vv` = vector-vector op, `*.sv` = scalar-vector op, `nthr` = number of μTs, `tidx` = current microthread's index.

simple way to map complex data-dependent control flow with μT scalar branches (inst. 4). If the μTs diverge at a branch, the VIU uses internally generated masks to disable inactive μTs along each path. The NVIDIA Fermi graphics processor [16] exemplifies this pattern with 32 multithreaded SIMT cores each with 16 lanes.

The **VT** pattern is also a hybrid but takes a very different approach from SIMT. As shown in Figure 2(e), the HT manages a collection of CTs, and each CT in turn manages an array of μTs. Figure 3(d) shows example VT assembly code. Like vector-SIMD, the CT can amortize control overheads and execute efficient vector memory instructions. Unlike vector-SIMD, the CT can use a *vector-fetch instruction* (inst. 8) to indicate the start of a scalar instruction stream that should be executed by the μTs. Explicit stop instructions (inst. 22) indicate a μT has finished the vector-fetched stream, and all μTs reconverge at the next vector fetch. As in SIMT, the VT VIU will try to execute across the μTs in a SIMD manner, but a vector-fetched scalar branch (inst. 16) can cause the μTs to diverge. Maven, introduced in this paper, and the earlier Scale [10] processor are examples of VT.

3. MICROARCHITECTURE OF MIMD, VECTOR-SIMD, AND VT TILES

In this section, we describe in detail the microarchitectures used to evaluate the various patterns. A data-parallel accelerator will usually include an array of tiles and an on-chip network to connect them to each other and an outer-level memory system, as shown in Figure 4(a). Each tile includes one or more tightly coupled cores and their caches, with examples in Figure 4(b)–(d). In this paper, we focus on comparing the various architectural design patterns with respect to a single data-parallel tile. The inter-tile interconnect and memory system are also critical components of a DLP accelerator system, but are outside the scope of this work.

3.1 Microarchitectural Components

We developed a library of parameterized synthesizable RTL components that can be combined to construct MIMD, vector-SIMD and VT tiles. Our library includes long-latency functional units, a multi-threaded scalar integer core, vector lanes, vector memory units, vector issue units, and blocking and non-blocking caches.

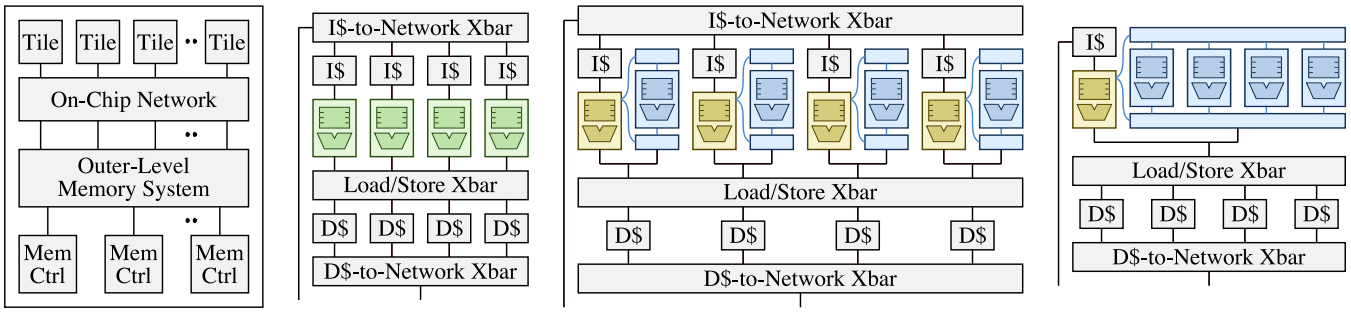
A set of **long-latency functional units** provide support for integer multiplication and division, and IEEE single-precision floating-point addition, multiplication, division, and square root. These units can be flexibly retimed to meet various cycle-time constraints.

Our **scalar integer core** implements a RISC ISA, with basic integer instructions executed in a five-stage, in-order pipeline but with two sets of request/response queues for attaching the core to the memory system and long-latency functional units. A two-read-port/two-write-port (2r2w-port) 32-entry 32-bit regfile holds both integer and floating-point values. One write port is for the integer pipeline and the other is shared by the memory system and long-latency functional units. The core can be multithreaded, with replicated architectural state for each thread and a dynamic thread scheduling stage at the front of the pipeline.

Figure 5 shows the microarchitectural template used for all the vector-based cores. A control processor (CP) sends vector instructions to the vector unit, which includes one or more vector lanes, a vector memory unit (VMU), and a vector issue unit (VIU). The lane and VMU components are nearly identical in all of the vector-based cores, but the VIU differs significantly between the vector-SIMD and VT cores as discussed below.

Our baseline **vector lane** consists of a unified 6r3w-port vector regfile and five vector functional units (VFUs): two arithmetic units (VAUs), a load unit (VLU), a store unit (VSU), and an address-generation unit (VGU). Each VAU contains an integer ALU and a subset of the long-latency functional units. The vector regfile can be dynamically reconfigured to support between 4–32 registers per μT with corresponding changes in maximum vector length (32–1). Each VFU has a sequencer to step through elements of each vector operation, generating physical register addresses.

The **vector memory unit** coordinates data movement between the memory system and the vector regfile using decoupling [5]. The CP splits each vector memory instruction into a vector memory μop issued to the VMU and a vector register access μop sent to the VIU, which is eventually issued to the VLU or VSU in the vector lane. A load μop causes the VMU to issue a vector's worth of load requests



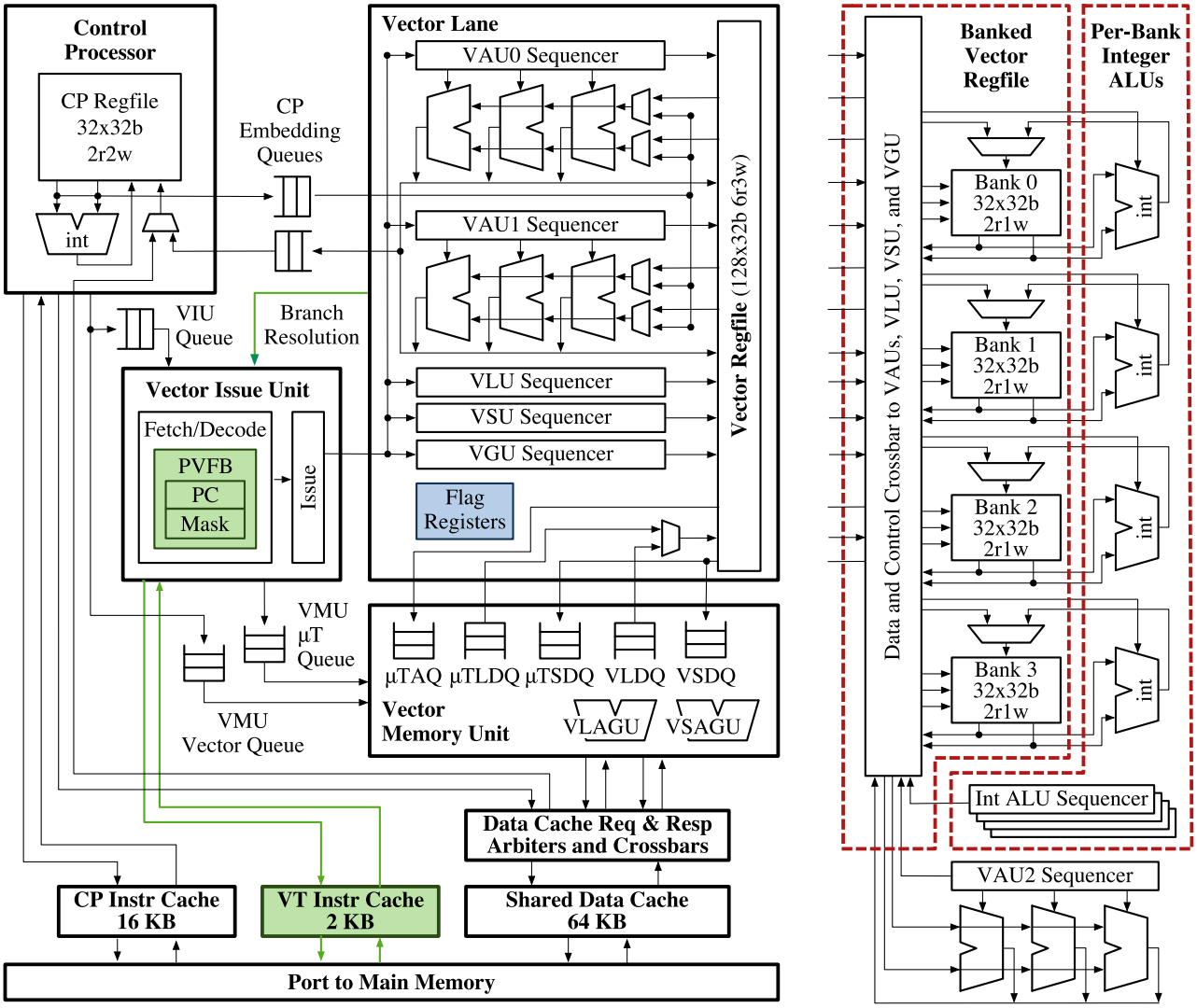
(a) Data-Parallel Accelerator

(b) MIMD Tile with Four Cores

(c) Vector-SIMD Tile with Four Single-Lane Cores

(d) Vector-SIMD Tile with One Four-Lane Core

Figure 4: Example Data-Parallel Tile Configurations



(a) Baseline Vector-SIMD and VT Core Microarchitecture

(b) Banked Regfile w/ Per-Bank Int ALUs

Figure 5: Vector-Based Core Microarchitecture – (a) Each vector-based core includes one or more vector lanes, vector memory unit, and vector issue unit; PVFB = pending vector fragment buffer, PC = program counter, VAU = vector arithmetic unit, VLU = vector load-data writeback unit, VSU = vector store-data read unit, VGU = address generation unit for µT loads/stores, VLDQ = vector load-data queue, VSDQ = vector store-data queue, VLAGU/VSAGU = address generation unit for vector loads/stores, µTAQ = µT address queue, µTLDQ = µT load-data queue, µTSDQ = µT store-data queue. Modules specific to vector-SIMD or VT cores are highlighted. (b) Changes required to implement intra-lane vector regfile banking with per-bank integer ALUs.

to the memory system, with data returned to the vector load data queue (VLDQ). As data becomes available, the VLU copies it from the VLDQ to the vector regfile. A store μop causes the VMU to retrieve a vector's worth of data from the vector store data queue (VSDQ) as it is pushed onto the queue by the VSU. Note that for single-lane configurations, the VMU still uses wide accesses between the VLDQ/VSDQ and the memory system, but moves data between the VLDQ/VSDQ and the vector lane one element at a time. Individual μT loads and stores (gathers and scatters) are handled similarly, except addresses are generated by the VGU and data flows through separate queues.

The main difference between vector-SIMD and VT cores is how the **vector issue unit** fetches instructions and handles conditional control flow. In a vector-SIMD core, the CP sends individual vector instructions to the VIU, which is responsible for ensuring that all hazards have been resolved before sending vector μops to the vector lane. Our vector-SIMD ISA supports data-dependent control flow using conventional vector masking, with eight single-bit flag registers. A μT is prevented from writing results for a vector instruction when the associated bit in a selected flag register is clear.

In our VT core, the CP sends vector-fetch instructions to the VIU. For each vector fetch, the VIU creates a new *vector fragment* consisting of a program counter, initialized to the start address specified in the vector fetch, and an active μT bit mask, initialized to all active. The VIU then fetches and executes the corresponding sequential instruction stream across all active μTs, sending a vector μop plus active μT mask to the vector lane for each instruction. The VIU handles a branch instruction by issuing a compare μop to one of the VFUs, which then produces a branch-resolution bit mask. If the mask is all zeros or ones, the VIU continues fetching scalar instructions along the fall-through or taken path. Otherwise, the μTs have diverged and so the VIU splits the current fragment into two fragments representing the μTs on the fall-through and taken paths, and continues to execute the fall-through fragment while placing the taken fragment in a *pending vector fragment buffer* (PVFB). The μTs can repeatedly diverge, creating new fragments, until there is only one μT per fragment. The current fragment finishes when it executes a `stop` instruction. The VIU then selects another vector fragment from the PVFB for execution. Once the PVFB is empty, indicating that all the μTs have stopped executing, the VIU can begin processing the next vector-fetch instruction.

Our library also includes **blocking and non-blocking cache** components with a rich set of parameters: cache type (instruction/data), access port width, refill port width, cache line size, total capacity, and associativity. For non-blocking caches, additional parameters include the number of miss-status-handling registers (MSHR) and the number of secondary misses per MSHR.

3.2 Constructing Tiles

MIMD cores combine a scalar integer core with integer and floating-point long-latency functional units, and support from one to eight μTs per core. **Vector cores** use a single-threaded scalar integer core as the CP connected to either a vector-SIMD or VT VIU, with one or more vector lanes and a VMU. To save area, the CP shares long-latency functional units with the vector lane, as in the Cray-1 [19].

We constructed two tile types: **multi-core tiles** consist of four MIMD (Figure 4(b)) or single-lane vector cores (Figure 4(c)), while **multi-lane tiles** have a single CP connected to a four-lane vector unit (Figure 4(d)). All tiles have the same number of long-latency functional units. Each tile includes a shared 64-KB four-bank data cache (8-way set-associative, 8 MSHRs, 4 secondary misses per MSHR), interleaved by 64-byte cache line. Request and response arbiters and crossbars manage communication between the cache banks and cores (or lanes). Each CP has a 16-KB private instruction cache and each VT VIU has a 2-KB vector instruction cache. Hence the overall instruction cache capacity (and area) is much larger in multi-core (64–72 KB) as compared to multi-lane (16–18 KB) tiles.

3.3 Microarchitectural Optimizations

We explored a series of microarchitectural optimizations to improve performance, area, and energy efficiency of our baseline vector-SIMD and VT cores. The first was using a conventional **banked vector register file** to reduce area and energy (see Figure 5(b)). While a monolithic 6r3w regfile simplifies vector lane design by allowing each VFU to access any element on any clock cycle, the high port count is expensive. Dividing the regfile into four independent banks each with one write and two read ports significantly reduces regfile area while keeping capacity constant. A crossbar connects banks to VFUs. Registers within a μT are co-located within a bank, and μTs are striped across banks. As a VFU sequencer iterates through the μTs in a vector, it accesses a new bank on each clock cycle. The VIU must schedule vector μops to prevent bank conflicts, where two VFUs try to access the same bank on the same clock cycle. The four 2r1w banks result in a greater aggregate bandwidth of eight read and four write ports, which we take advantage of by adding a third VAU (VAU2) to the vector lane and rearranging the assignment of functional units to VAUs.

We developed another optimization for the banked design, which removes integer units from the VAUs and instead adds four **per-bank integer ALUs** directly connected to the read and write ports of each bank, bypassing the crossbar (see Figure 5(b)). This saves energy, and also helps performance by avoiding structural hazards and increasing peak integer throughput to four integer VAUs. The area cost of the extra ALUs is small relative to the size of the regfile.

We also investigated **density-time execution** [21] to improve vector performance on irregular codes. The baseline vector machine takes time proportional to the vector length for each vector instruction, regardless of the number of inactive μTs. Codes with highly irregular control flow often cause significant divergence between the μTs, splintering a vector into many fragments of only a few active μTs each. Density-time improves vector execution efficiency by "compressing" the vector fragment and only spending cycles on active μTs. Bank scheduling constraints reduce the effectiveness of density-time execution in banked regfiles. Multi-lane machines have even greater constraints, as lanes must remain synchronized, so we only added density-time to single-lane machines.

The PVFB in our baseline VT machine is a FIFO queue with no means to merge vector fragments. Hence once a vector becomes fragmented, those fragments will execute independently until all μTs execute a `stop` instruction, even when fragments have the same PC. We developed two new schemes for VT machines to implement **dynamic fragment convergence** in the PVFB. When a new fragment is pushed into the PVFB, both schemes will attempt to dynamically merge the fragment with an existing fragment if their PCs match, OR-ing their active μT masks together. The challenge is to construct a fragment scheduling heuristic that maximizes opportunities for convergence by avoiding executing a fragment if it could later merge with another fragment in the PVFB. Note the Maven VT design uses the same scalar ISA for both the CP and the vector μTs, with no explicit static hints to aid fragment convergence as are believed to be used in SIMT architectures [16].

Configuration	Num Cores	Per Core: Num Regs	Per Core: Num µTs	Peak Throughput: Arith (ops/cyc)	Peak Throughput: Mem (elm/cyc)	Power: Statistical (mW)	Power: Simulated (mW)	Total Area (mm^2)	Cycle Time (ns)
mimd-c4r32§	4	32	4	4	4	149	137 – 181	3.7	1.10
mimd-c4r64§	4	64	8	4	4	216	130 – 247	4.0	1.13
mimd-c4r128§	4	128	16	4	4	242	124 – 261	4.2	1.19
mimd-c4r256§	4	256	32	4	4	299	221 – 298	4.7	1.27

Configuration	Num Cores	Per Core: Num Lanes	Per Core: Max vlen Range	Per Lane: Num Regs	Peak Throughput: Arith (ops/cyc)	Peak Throughput: Mem (elm/cyc)	Power: Statistical (mW)	Power: Simulated (mW)	Total Area (mm^2)	Cycle Time (ns)
vsimd-c4v1r256+bi§	4	1	8 – 32	256	4c + 16v	4l + 4s	396	213 – 331	5.6	1.37
vsimd-c1v4r256+bi§	1	4	32 – 128	256	1c + 16v	4l + 4s	224	137 – 252	3.9	1.46
vt-c4v1r256	4	1	8 – 32	256	4c + 8v	4l + 4s	428	162 – 318	6.3	1.47
vt-c4v1r256+b	4	1	8 – 32	256	4c + 8v	4l + 4s	404	147 – 271	5.6	1.31
vt-c4v1r256+bi	4	1	8 – 32	256	4c + 16v	4l + 4s	445	172 – 298	5.9	1.32
vt-c4v1r256+bi+2s	4	1	8 – 32	256	4c + 16v	4l + 4s	409	225 – 304	5.9	1.32
vt-c4v1r256+bi+2s+d§	4	1	8 – 32	256	4c + 16v	4l + 4s	410	168 – 300	5.9	1.36
vt-c1v4r256+bi+2s§	1	4	32	256	1c + 16v	4l + 4s	205	111 – 167	3.9	1.42
vt-c1v4r256+bi+2s+mc	1	4	32	256	1c + 16v	4l + 4s	223	118 – 173	4.0	1.42

Table 1: Subset of Evaluated Tile Configurations – Multi-core and multi-lane tiles for MIMD, vector-SIMD, and VT patterns. Configurations with § are used in Section 5.3. *statistical power* column is from post-PAR; *simulated power* column shows min/max across all gate-level simulations; *configuration* column: b = banked, bi = banked+int, 2s = 2-stack, d = density-time, mc = memory coalescing; *num µTs* column is the number of µTs supported with the default of 32 registers/µT; *arith* column: xc + yv = x CP ops and y vector unit ops per cycle; *mem* column: xl + ys = x load elements and y store elements per cycle.

Our first convergence scheme, called *1-stack*, organizes the PVFB as a stack with fragments sorted by PC address, with newly created fragments systolically insertion-sorted into the stack. The VIU always selects the PVFB fragment with the numerically smallest PC as the next to execute. The intuition behind 1-stack is to favor fragments trailing behind in execution, giving them more chance to meet up with faster-moving fragments at a convergence point.

The 1-stack scheme performs reasonably well, but is sub-optimal for loops with multiple backwards branches, as fragments which first branch back for another loop iteration are treated as if they were behind slower fragments in the same iteration and race ahead. Our second scheme, called *2-stack*, divides the PVFB into two virtual stacks, one for fragments on the current iteration of a loop and another for fragments on a future iteration of a loop. Fragments created from backwards branches are pushed onto the future stack, while the VIU only pops fragments from the current stack. When the current stack empties, the current and future stacks are swapped.

The final optimization we considered is a **dynamic memory coalescer** for multi-lane VT vector units. During the execution of a µT load or store instruction, each lane can generate a separate memory address on each cycle. The memory coalescer looks across lanes for opportunities to satisfy multiple µT accesses from a single 128-bit memory access. This can significantly help performance on codes that use µT loads and stores to access memory addresses with a unit stride, as these would otherwise generate cache bank conflicts.

4. EVALUATION FRAMEWORK

This section describes the hardware and software infrastructure used to evaluate the various microarchitectural options introduced in the previous section, and also outlines the specific configurations, microbenchmarks, and application kernels used in our evaluation.

4.1 Hardware Toolflow

We use our own machine definition files to instantiate and compose the parameterized Verilog RTL into a full model for each tile configuration. We targeted TSMC's 65-nm GPLUSTC process using a Synposys-based ASIC toolflow: VCS for simulation, Design Compiler for synthesis, and IC Compiler for place-and-route (PAR). RTL simulation produces cycle counts. PAR produces cycle time and area estimates. Table 1 lists IC Compiler post-PAR power estimates based on a uniform statistical probability of bit transitions, and the range of powers reported via PrimeTime across all benchmarks when using bit-accurate activity for every net simulated on a back-annotated post-PAR gate-level model. The inaccuracy of the IC Compiler estimates and the large variance in power across benchmarks, motivated us to use only detailed gate-level simulation energy results from PrimeTime.

Complex functional units (e.g., floating-point) are implemented using Synopsys DesignWare library components, with automatic register retiming to generate pipelined units satisfying our cycle-time constraint. The resulting latencies were: integer multiplier (3) and divider (12), floating-point adder (3), multiplier (3), divider (7), and square-root unit (10).

We did not have access to a memory compiler for our target process, so we model SRAMs and caches by creating abstracted "black-box" modules, with area, timing, and power models suitable for use by the CAD tools. We used CACTI [14] to explore a range of possible implementations and chose one that satisfied our cycle-time requirement while consuming minimal power and area. We compared CACTI's predicted parameter values to the SRAM datasheet for our target process and found them to be reasonably close. Cache behavior is modeled by a cache simulator (written in

		Control Thread			Microthread										Active µT Distribution (%)			
	Name	vf	vec ld	vec st	int	fp	ld	st	amo	br	cmv	tot	loop	nregs	1–25	26–50	51–75	76–100
µbmarks	vvadd	1	2u	2u		1						2		4				100.0
	bsearch-cmv	1	1u	1u	17		2			1	4	25	×	13	1.0	3.3	5.8	89.9
	bsearch	1	1u	1u	15		3			5	1	26	×	10	77.6	12.4	5.1	4.8
	bsearch (w/ 1-stack)														23.8	23.4	11.7	41.0
	bsearch (w/ 2-stack)														10.1	26.8	49.2	13.9
App Kernels	viterbi	3	3u	1u, 4s	21						3	35		8				100.0
	rsort	3	3u, 2s	3u	14		2	3	1			25		11				100.0
	kmeans	9	7u, 3s	5u, 1s	12	6	2	2	1	1	2	40		8				100.0
	dither	1	4u, 1s	5u, 1s	13					1	2	24		8	0.2	0.4	0.7	98.7
	physics	4	6u, 12s	1u, 9s	5	56	24	4		16		132	×	32	6.9	15.0	28.7	49.3
	physics (w/ 2-stack)														4.7	13.1	28.3	53.9
	strsearch	3	5u	1u	35		9	5		15	2	96	×	14	57.5	25.5	16.9	0.1
	strsearch (w/ 2-stack)														14.8	30.5	54.7	0.1

Table 2: Microbenchmark and Application Kernel Statistics for VT Implementation – Number of instructions listed by type. Distribution of active µTs with a FIFO PVFB unless otherwise specified in *name* column. Each section sorted from most regular to most irregular. *vec ld/st* columns indicate numbers of both unit-stride (u) and strided (s) accesses; *loop* column indicates an inner loop within the vector-fetched block; *nregs* column indicates number of registers a vector-fetched block requires.

C++) that interfaces with the ports of the cache modules. The latency between a cache-line refill request and response was set at 50 cycles. We specify the dimensions of the target ASIC and the placement and orientation of the large black-box modules. The rest of the design (including regfiles) was implemented using standard cells, all automatically placed.

4.2 Tile Configurations

We evaluated hundreds of tile configurations using our hardware toolflow. For this paper, we focus on 22 representative configurations; Table 1 lists 13 of these, with the remaining 9 introduced later in the paper. We name configurations beginning with a prefix designating the style of machine, followed by the number of cores (c), the number of lanes (v), and physical registers (r) per core or lane. The suffix denotes various microarchitectural optimizations: b = banked regfile, bi = banked regfile with extra integer ALUs, 1s = 1-stack convergence scheme, 2s = 2-stack convergence scheme, d = density-time execution, mc = memory coalescing. Each type of core is implemented with 32, 64, 128, and 256 physical registers. For the MIMD cores, this corresponds to 1, 2, 4, and 8 µTs respectively. For the vector cores, the maximum hardware vector length is determined by the size of the vector regfile and the number of registers assigned to each µT (4–32). The vector length is capped at 32 for all VT designs, even though some configurations (i.e., 256 physical registers with 4 registers per µT) could theoretically support longer vector lengths. We imposed this limitation because some structures in the VT machines (such as the PVFB) scale quadratically in area with respect to the maximum number of active µTs. Banked vector regfile designs are only implemented for 128 and 256 physical registers.

4.3 Microbenchmarks & Application Kernels

We selected several microbenchmarks and six larger application kernels to represent the spectrum from regular to irregular DLP.

The *vvadd* microbenchmark performs a 1000-element vector-vector floating-point addition and is the simplest example of regular DLP. The *bsearch* microbenchmark uses a binary search algorithm to perform 1000 look-ups into a sorted array of 1000 key-value pairs. This microbenchmark exhibits highly irregular DLP with two nested loops: an outer `for` loop over the search keys and an inner `while` loop implementing a binary search for finding the key. We include two VT implementations: one (*bsearch*) uses branches to handle intra-iteration control flow, while the second (*bsearch-cmv*) uses conditional moves explicitly inserted by the programmer.

The *viterbi* kernel decodes frames of convolutionally encoded data using the Viterbi algorithm. Iterative calculation of survivor paths and their accumulated error are parallelized across paths. Each µT performs an add-compare-select butterfly operation to compute the error for two paths simultaneously, which requires unpredictable accesses to a lookup table. The *rsort* kernel performs an incremental radix sort on an array of integers. During each iteration, individual µTs build local histograms of the data, and then a parallel reduction is performed to determine the mapping to a global destination array. Atomic memory operations are necessary to build the global histogram structure. The *kmeans* kernel implements the k-means clustering algorithm. It classifies a collection of objects, each with some number of features, into a set of clusters through an iterative process. Assignment of objects to clusters is parallelized across objects. The minimum distance between an object and each cluster is computed independently by each µT and an atomic memory operation updates a shared data structure. Cluster centers are recomputed in parallel using one µT per cluster. The *dither* kernel generates a black and white image from a gray-scale image using Floyd-Steinberg dithering. Work is parallelized across the diagonals of the image, so that each µT works on a subset of the diagonal. A data-dependent conditional allows µTs to skip work if an input pixel is white. The *physics* kernel performs a simple Newtonian physics simulation with object collision detection. Each µT is responsible for intersection detection, motion variable computation, and location calculation for a single object. Oct-trees are also generated in parallel. The *strsearch* kernel implements the Knuth-Morris-Pratt algorithm to search a collection of byte streams for the presence of substrings. The search is parallelized by having all µTs search for the same substrings in different streams. The DFAs used to model substring-matching state machines are also generated in parallel.

```
void idlp( int c[], int a[], int b[], int n, int x ) {
  int vlen = vt::config( 7, n ); // config vector unit
  vt::HardwareVector<int> vx(x);
  for ( int i = 0; i < n; i += vlen ) {
    vlen = vt::set_vlen(n-i); // stripmining

    vt::HardwareVector<int*> vcptr(&c[i]);
    vt::HardwareVector<int> va, vb;
    va.load(&a[i]); // unit-stride vector load
    vb.load(&b[i]); // unit-stride vector load

    VT_VFETCH( (vcptr,vx,va,vb), ({
      if ( va > 0 )
        vcptr[vt::get_utidx()] = vx * va + vb;
    }));
  }
  vt::sync_cv(); // vector memory fence
}
```

Figure 6: Irregular DLP Example Using VT C++ Library – Code for loop in Figure 1(d). Roughly compiles to assembly in Figure 3(d). `config()` specifies number of required μT registers. `set_vlen()` sets number of active μTs. `get_utidx()` returns μT's thread index. `HardwareVector<T>` type enables moving data in and out of vector registers; compiler handles vector register allocation. `VT_VFETCH` macro expands to vector fetch for given code block. `HardwareVector<T>` objects act as vectors outside block and as scalars inside block. Any valid C++ is allowed inside a vector-fetched block excluding system calls and exceptions.

Table 2 reports the instruction counts and distribution of active μTs for the VT implementations of two representative microbenchmarks and the six application kernels. *viterbi* is an example of regular DLP with known memory access patterns. *rsort*, *kmeans*, and *dither* all exhibit mild control flow conditionals with more irregular memory access patterns. *physics* and *strsearch* exhibits characteristics of highly irregular DLP code: loops with data-dependent exit conditionals, highly irregular data access patterns, and many conditional branches.

4.4 Programming Methodology

Past accelerators usually relied on hand-coded assembly or compilers that automatically extract DLP from high-level programming languages [1,4,7]. Recently there has been a renewed interest in explicitly data-parallel programming methodologies [3,15,17], where the programmer writes code for the HT and annotates data-parallel tasks to be executed in parallel on all μTs. We developed a similar explicit-DLP C++ programming environment for Maven. We modified the GNU C++/newlib toolchain to generate code for the unified ISA used on both CT and μTs, and also developed a VT library to manage the interaction between the two types of thread (see Figure 6).

For vector-SIMD, we were able to leverage the built-in GCC vectorizer for mapping very simple regular DLP microbenchmarks, but the GCC vectorizer cannot automatically compile the larger application kernels for the vector-SIMD tiles. For these more complicated vector-SIMD kernels, we use a subset of our VT C++ library for stripmining and vector memory operations along with GCC's inline assembly extensions for the actual computation. We used a very similar vectorization approach as in the VT implementations, but the level of programmer effort required for vector-SIMD was substantially higher. Our struggle to find a suitable way to program more interesting codes for the vector-SIMD pattern is anecdotal evidence of the broader challenge of programming such accelerators, and this helped motivate our interest in the VT programming model.

MIMD code is written using a custom lightweight threading library, and applications explicitly manage thread scheduling. For all systems, a simple proxy kernel running on the cores supports basic system calls by communicating with an application server running on the host.

5. EVALUATION RESULTS

In this section, we first compare tile configurations based on their cycle time and area before exploring the impact of various microarchitectural optimizations. We then compare implementation efficiency and performance of the MIMD, vector-SIMD, and VT patterns for the six application kernels. We present highlights from our results here; more extensive results are available separately [11].

5.1 Cycle Time and Area Comparison

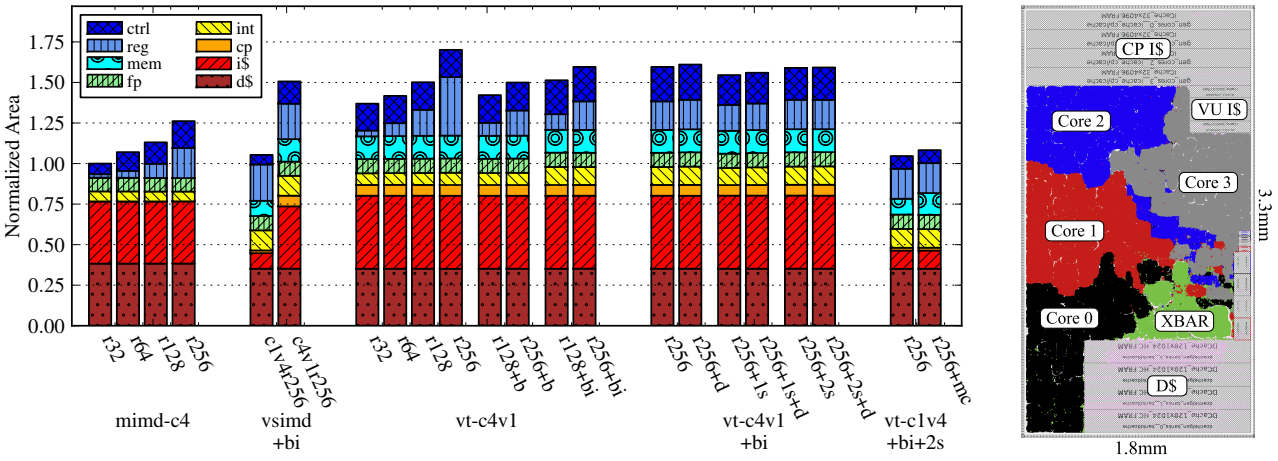
Tile cycle times vary from 1.10–1.47 ns (see Table 1), with critical paths usually passing through the crossbar that connects cores to individual data cache banks. Figure 7(a) shows the area breakdown of the tiles normalized to a *mimd-c4r32* tile. The caches contribute the most to the area of each tile. Note that a multi-core vector-SIMD tile (*vsimd-c4v1r256+bi*) is 20% larger than a multi-core MIMD tile with the same number of long-latency functional units and the same total number of physical registers (*mimd-c4r256*) due to the sophisticated VMU and the extra integer ALUs per bank. A multi-lane vector-SIMD tile (*vsimd-c1v4r256+bi*) is actually 16% smaller than the *mimd-c4r256* tile because the increased area overheads are amortized across four lanes. Note that we added additional buffer space to the multi-lane tiles to balance the performance across vector tiles, resulting in similar area usage of the memory unit for both multi-core and multi-lane vector tiles. Across all vector tiles, the overhead of the embedded control processor is less than 5%, since it shares long-latency functional units with the vector unit.

Comparing a multi-core VT tile (*vt-c4v1r256+bi*) to a multi-core vector-SIMD tile (*vsimd-c4v1r256+bi*) shows the area overhead of the extra VT mechanisms is only ≈6%. The VT tile includes a PVFB instead of a vector flag regfile, causing the regfile area to decrease and the control area to increase. There is also a small area overhead due to the extra VT instruction cache. For multi-lane tiles, these VT overheads are amortized across four lanes making them negligible (compare *vt-c1v4r256+bi+2s* vs. *vsimd-c1v4r256+bi*).

5.2 Microarchitectural Tradeoffs

Figure 8 shows the impact of increasing the number of physical registers per core or lane when executing *bsearch-cmv*. For *mimd-c4r**, increasing the number of μTs from 1 to 2 improves performance but at an energy cost. The energy increase is due to a larger regfile (now 64 registers per core) and more control overhead. Supporting more than two μTs reduces performance due to the non-trivial start-up overhead required to spawn and join the additional μTs and a longer cycle time. In the *vt-c4v1* tile with a unified vector regfile, adding more vector register elements increases hardware vector length and improves temporal amortization of the CP, instruction cache, and control energy. At 256 registers, however, the larger access energy of the unified regfile outweighs the benefits of increased vector length. The performance also decreases since the access time of the regfile becomes critical.

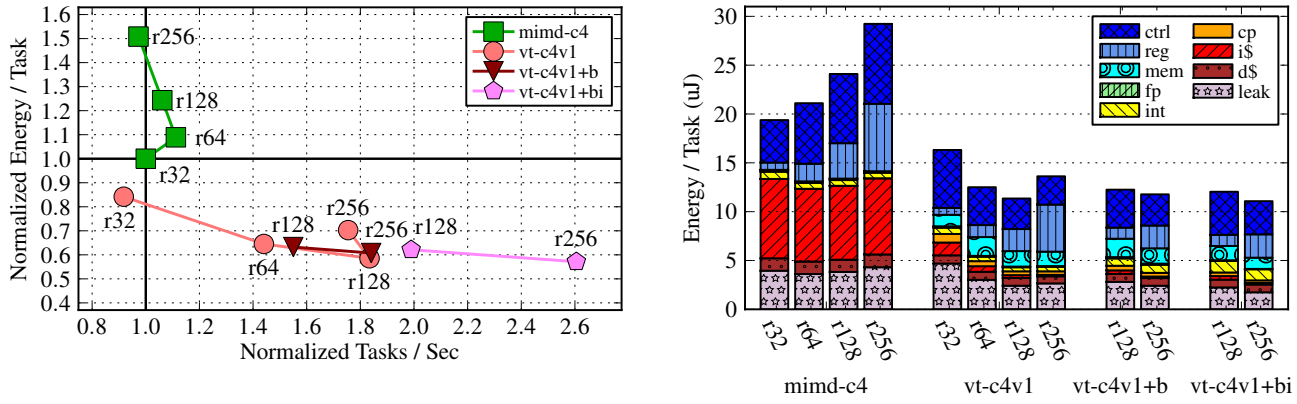
Figure 8 also shows the impact of regfile banking and adding per-bank integer ALUs. Banking a regfile with 128 entries reduces regfile access energy but decreases performance due to bank conflicts



(a) Area Breakdown for Evaluated Tile Configurations

(b) ASIC Layout for *vt-c4v1r256+bi+2s+d*

Figure 7: Area and VLSI Layout for Tile Configurations – (a) area breakdown for each of the 22 tile configurations normalized to the *mimd-c4r32* tile, (b) ASIC layout for *vt-c4v1r256+bi+2s+d* with individual cores and memory crossbar highlighted.



(a) Energy vs. Performance for *bsearch-cmv*

(b) Energy Breakdown for *bsearch-cmv*

Figure 8: Impact of Additional Physical Registers, Intra-Lane Regfile Banking, and Additional Per-Bank Integer ALUs – Results for multi-core MIMD and VT tiles running the *bsearch-cmv* microbenchmark.

(see *vt-c4v1+b* configuration). Adding per-bank integer ALUs partially offsets this performance loss (see *vt-c4v1+bi* configuration). With the additional ALUs, a VT tile with a banked regfile improves both performance and energy versus a VT tile with a unified regfile. Figure 7(a) shows that banking the vector regfile reduces the regfile area by a factor of 2×, while adding local integer ALUs in a banked design only modestly increases the integer and control logic area. Based on analyzing results across many tile configurations and applications, we determined that banking the vector regfile and adding per-bank integer ALUs was the optimal choice for all vector tiles.

Figure 9 shows the impact of adding density-time execution and dynamic fragment convergence to a multi-core VT tile running *bsearch*. Adding just density-time execution eliminates significant wasted work after divergence, improving performance by 2.5× and reducing energy by 2×. Density-time execution is less useful on multi-lane configurations due to the additional constraints required for compression. Our stack-based convergence schemes are a different way of mitigating divergence by converging µTs when possible. For *bsearch*, the 2-stack PVFB forces µTs to stay on the same loop iteration, improving performance by 6× and reducing

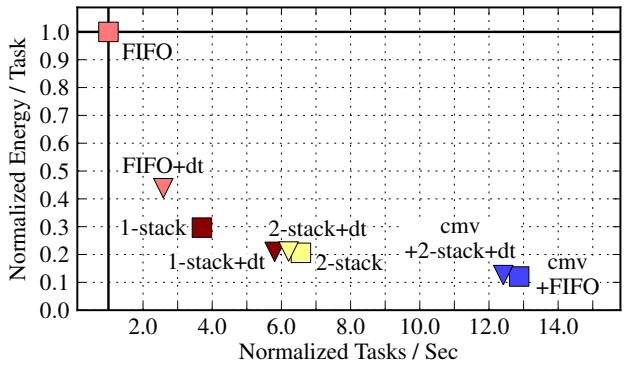


Figure 9: Impact of Density-Time Execution and Stack-Based Convergence Schemes – Results for multi-core VT tile running *bsearch* and *bsearch-cmv*.

energy by 5× as compared to the baseline FIFO PVFB. Combining density-time and a 2-stack PVFB has little impact here as the 2-stack scheme already removes most divergence (see Table 2). Our experience with other microbenchmarks and application ker-

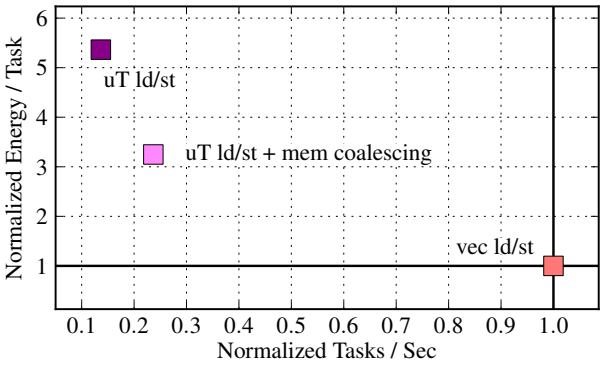


Figure 10: Impact of Memory Coalescing – Results for multi-lane VT tile running *vvadd*.

nels suggest that for codes where convergence is simply not possible the addition of density-time execution can have significant impact. Note that replacing branches with explicit conditional moves (*bsearch-cmv*) performs better than dynamic optimizations for µT branches, but µT branches are more general and simpler to program for irregular DLP codes. Table 1 and Figure 7(a) show that the 2-stack PVFB and density-time execution have little impact on area and cycle time. Based on our analysis, the 2-stack PVFB is used for both multi-core and multi-lane VT tiles, while density-time execution is only used on multi-core VT tiles.

Figure 10 illustrates the benefit of vector memory accesses versus µT memory accesses on a multi-lane VT tile running *vvadd*. Using µT memory accesses limits opportunities for access-execute decoupling and requires six additional µT instructions for address generation, resulting in over 5× worse energy and 7× worse performance for *vvadd*. Memory coalescing recoups some of the lost performance and energy efficiency, but is still far behind vector instructions. This small example hints at key differences between SIMT and VT. Current SIMT implementations use a very large number of µTs (and large regfiles) to hide memory latency instead of a decoupled control thread, and rely on dynamic coalescing instead of true vector memory instructions. However, exploiting these VT features requires software to factor out the common work from the µTs.

5.3 Application Kernel Results

Figure 11 compares the application kernel results between the MIMD, vector-SIMD, and VT tiles. The upper row plots overall energy/task against performance, while the lower row plots energy/task against area-normalized performance to indicate expected throughput from a given silicon budget for a highly parallel workload. Kernels are ordered to have increasing irregularity from left to right. We draw several broad insights from these results.

First, we observed that adding more µTs to a multi-core MIMD tile is not particularly effective, especially when area is considered. We found parallelization and load-balancing become more challenging for the complex application kernels, and adding µTs can hurt performance in some cases due to increased cycle time and non-trivial interactions with the memory system.

Second, we observed that the best vector-based machines are generally faster and/or more energy-efficient than the MIMD cores though normalizing for area reduces the relative advantage, and for some irregular codes the MIMD cores perform slightly better (e.g., *strsearch*) though at a greater energy cost.

Third, comparing vector-SIMD and VT on the first four kernels, we see VT is more efficient than vector-SIMD for both multi-core single-lane (*c4v1*) and single-core multi-lane (*c1v4*) design points. Note we used hand-optimized vector-SIMD code but compiled VT code for these four kernels. One reason VT performs better than vector-SIMD, particularly on multi-lane *viterbi* and *kmeans*, is that vector-fetch instructions more compactly encode work than vector instructions, reducing pressure on the VIU queue and allowing the CT to run ahead faster.

Fourth, comparing *c4v1* versus *c1v4* vector machines, we see that the multi-lane vector designs are generally more energy-efficient than multi-core vector designs as they amortize control overhead over more datapaths. Another advantage we observed for multi-lane machines was that we did not have to partition and load-balance work across multiple cores. Multi-core vector machines sometimes have a raw performance advantage over multi-lane vector machines. Our multi-lane tiles have less address bandwidth to the shared data cache, making code with many vector loads and stores perform worse (*kmeans* and *physics*). Lack of density-time execution and no ability to run independent control threads also reduces efficiency of multi-lane machines on irregular DLP code. However, these performance advantages for multi-core vector machines usually disappear once area is considered, except for the most irregular kernel *strsearch*. The area difference is mostly due to the disparity in aggregate instruction cache capacity.

Overall, our results suggest a single-core multi-lane VT tile with the 2-stack PVFB and a banked regfile with per-bank integer ALUs (*vt-c1v4r256+bi+2s*) is a good design point for Maven.

6. CONCLUSIONS

Effective data-parallel accelerators must handle regular and irregular DLP efficiently and still retain programmability. Our detailed VLSI results confirm that vector-based microarchitectures are more area and energy efficient than scalar-based microarchitectures, even for fairly irregular data-level parallelism. We introduced Maven, a new simpler vector-thread microarchitecture based on the traditional vector-SIMD microarchitecture, and showed that it is superior to traditional vector-SIMD architectures by providing both greater efficiency and easier programmability. Maven's efficiency is improved with several new microarchitectural optimizations, including efficient dynamic convergence for microthreads and ALUs distributed close to the banks within a banked vector register file.

In future work, we are interested in a more detailed comparison of VT to the popular SIMT design pattern. Our initial results suggest that SIMT will be less efficient though easier to program than VT. We are also interested in exploring whether programming environment improvements can simplify the programming of vector-SIMD machines to reduce the need for VT or SIMT mechanisms, and whether hybrid machines containing both pure MIMD and pure SIMD might be more efficient than attempting to execute very irregular code on SIMD hardware.

ACKNOWLEDGMENTS

This work was supported in part by Microsoft (Award #024263) and Intel (Award #024894, equipment donations) funding and by matching funding from U.C. Discovery (Award #DIG07-10227). The authors acknowledge and thank Jiongjia Fang and Ji Kim for their help writing application kernels, Christopher Celio for his help writing Maven software and developing the vector-SIMD instruction set, and Hidetaka Aoki for his early feedback on the Maven microarchitecture.

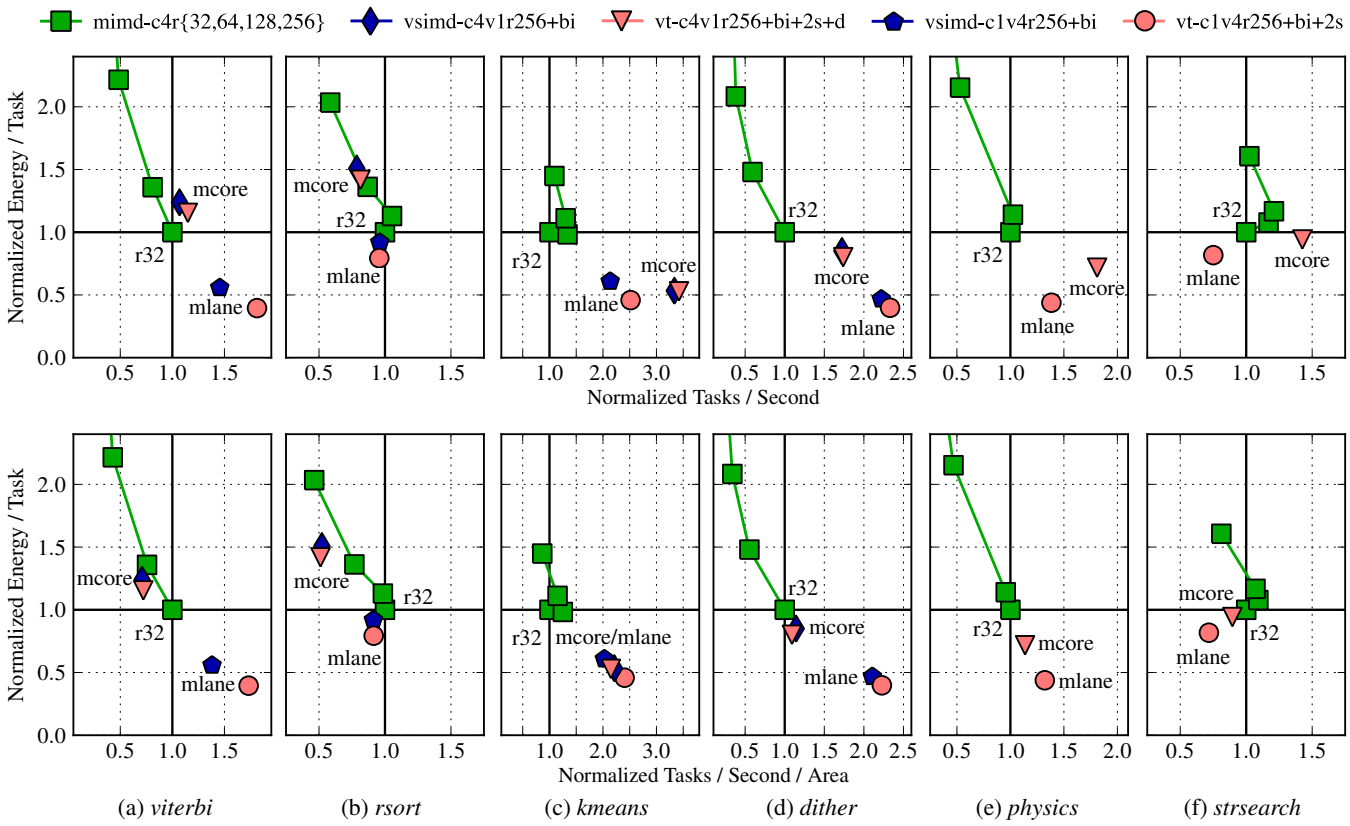


Figure 11: Implementation Efficiency and Performance for MIMD, vector-SIMD, and VT Patterns Running Application Kernels – Each column is for different kernel. Legend at top. *mimd-c4r256* is significantly worse and lies outside the axes for some graphs. There are no vector-SIMD implementations for *strsearch* and *physics* due to difficulty of implementing complex irregular DLP in hand-coded assembly. mcore = multi-core vector-SIMD/VT tiles, mlane = multi-lane vector-SIMD/VT tiles, r32 = MIMD tile with 32 registers (i.e., one µT).

REFERENCES

[1] D. F. Bacon et al. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, Dec 1994.

[2] C. Batten. Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators. PhD Thesis, MIT, 2010.

[3] I. Buck et al. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug 2004.

[4] D. DeVries and C. G. Lee. A Vectorizing SUIF Compiler. *SUIF Compiler Workshop*, Jan 1995.

[5] R. Espasa and M. Valero. Decoupled Vector Architectures. *HPCA*, Feb 1996.

[6] M. Gschwind et al. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar 2006.

[7] M. Hampton and K. Asanović. Compiling for Vector-Thread Architectures. *CGO*, Apr 2008.

[8] J. H. Kelm et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *ISCA*, Jun 2009.

[9] R. Krashinsky. Vector-Thread Architecture and Implementation. PhD Thesis, MIT, 2007.

[10] R. Krashinsky et al. The Vector-Thread Architecture. *ISCA*, Jun 2004.

[11] Y. Lee. Efficient VLSI Implementations of Vector-Thread Architectures. MS Thesis, UC Berkeley, 2011.

[12] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[13] A. Mahesri et al. Tradeoffs in Designing Accelerator Architectures for Visual Computing. *MICRO*, Nov 2008.

[14] N. Muralimanohar et al. CACTI 6.0: A Tool to Model Large Caches, 2009.

[15] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar/Apr 2008.

[16] NVIDIA's Next Gen CUDA Compute Architecture: Fermi. NVIDIA White Paper, 2009.

[17] The OpenCL Specification. Khronos OpenCL Working Group, 2008.

[18] S. Rivoire et al. Vector Lane Threading. *Int'l Conf. on Parallel Processing*, Aug 2006.

[19] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[20] K. Sankaralingam et al. Universal Mechanisms for Data-Parallel Architectures. *MICRO*, Dec 2003.

[21] J. Smith et al. Vector Instruction Set Support for Conditional Operations. *ISCA*, Jun 2000.

[22] J. Wawrzynek et al. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, Mar 1996.